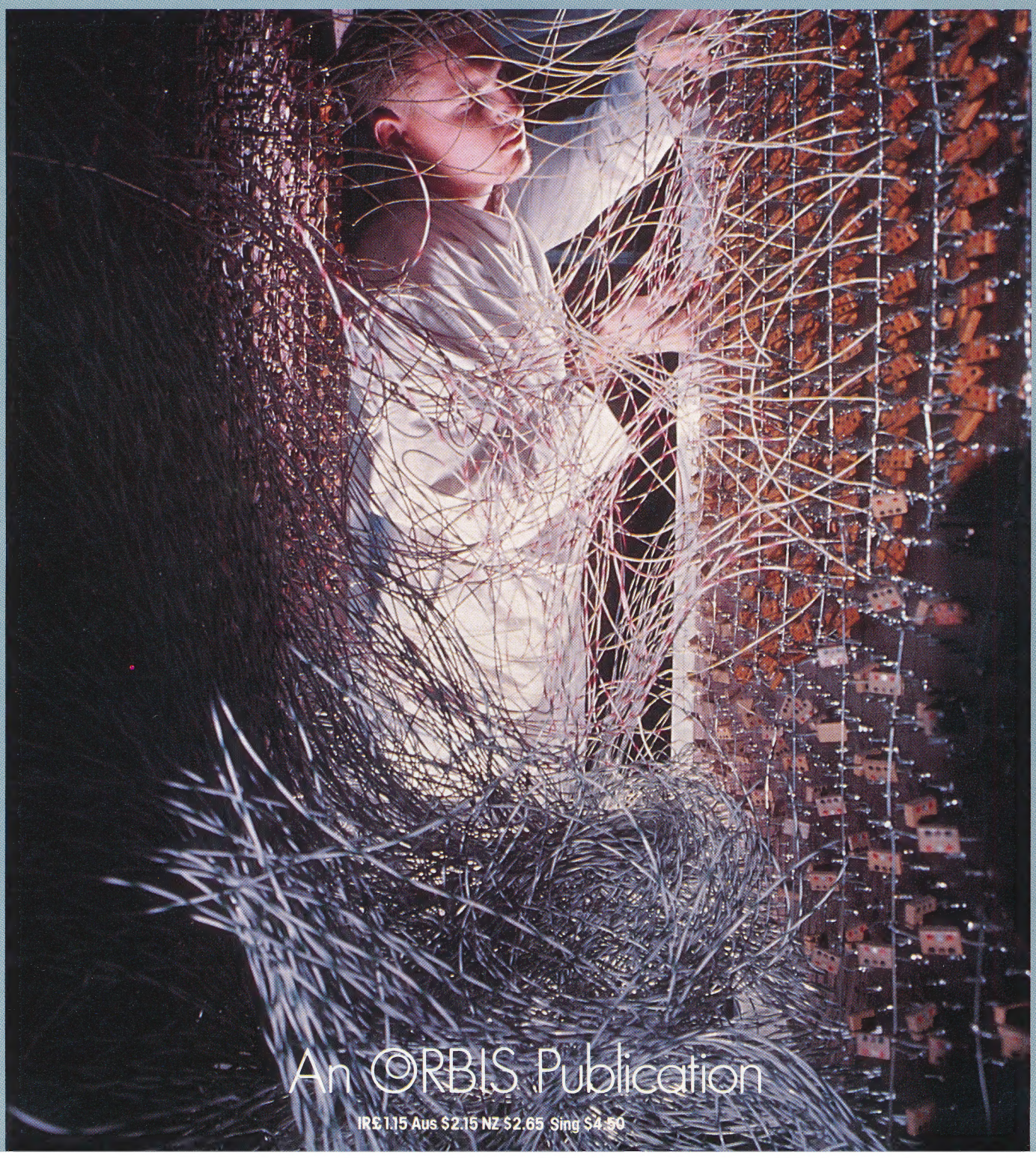
HUMPHRIES.

ISSN 0265-2919

90p 93

THE HOME COMPUTER ADVANCED COURSE

MAKING THE MOST OF YOUR MICRO



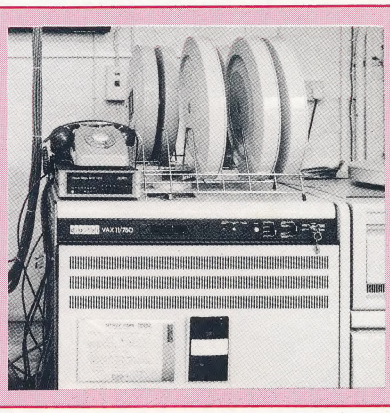
An ORBIS Publication

IR£1.15 Aus $2.15 NZ $2.65 Sing $4.50

CONTENTS

Next Week

- **The VAX minicomputer is where a great deal of micro software originates. We take a look inside the machine.**
- **Whizz-kids making millions from games written at home is one of the computing industry's great myths. We examine the reality of the games industry.**
- **We conclude our series on the C programming language.**



QUIZ

1) Does the traditional idea of parallel processing conform to von Neumann architecture?
2) What is a 'history dependent' program?
3) Why has virtual memory management come back into fashion?
4) In what way does I/O to files and devices differ under C?

Answers To Last Week's Quiz

1) The Hayes standard is a system of protocols and commands used on professional micros, especially those intended for the US market.
2) The link commands in 68000 assembly language are used to set aside an area of memory for the variables for a subroutine.
3) Clusters are units of multiple sectors on a disk.
4) A signal pattern tells the computer what kind of data is held in the following bits.

Coming Up

- A discussion of some of the developments expected in the microcomputer industry in forthcoming years.
- A look at the things to consider when choosing a computer language.

Editor Stephen Cooke; **Art Editor** Claudia Zeff; **Deputy Editor** Steve Colwill; **Production Editor** Bobby Pickering; **Designer** Julian Dorr; **Staff Writer** Steve Malone; **Art Assistant** Caroline Clayton; **Sub Editor** Jon Kaye; **Contributors** Mike Curtis, Steve Malone, David Fensome, Max Hotopf, Dick Pountain, Dave Nicholls; **Software Consultants** Pilot Software City; **Group Art Director** Perry Neville; **Managing Director** Stephen England; **Published by Orbis Publishing Ltd; Editorial Director** Brian Innes; **Project Development** Peter Brookesmith; **Executive Editor** Maurice Geller; **Production Assistant** Susan Brown; **Subscription Manager** Christine Allen; **Designed and produced by Bunch Partworks Ltd; Editorial Office** 14 Rathbone Place, London W1P 1DE;

Typeset by Universe; Reproduction by Mullis Morgan Ltd; Printed in Great Britain by Heanor Gate Printing Ltd, Derby

HOW TO OBTAIN ISSUES AND BINDERS FOR THE HOME COMPUTER ADVANCED COURSE – Issues can be obtained by placing an order with your newsagent or direct from our subscription department. If you have any difficulty obtaining any back issues from your newsagent, please write to us stating the issue(s) required and enclosing a cheque for the cover price of the issue(s). **AUSTRALIA** – please write to: Gordon & Gotch (Aus) Ltd, 114 William Street, PO Box 767G, Melbourne, Victoria 3001. **MALTA, NEW ZEALAND & SOUTH AFRICA –** Back numbers are available at cover price from your newsagent. In case of difficulty, write to the address given for binders.
UK/EIRE – Issue Price: 90p/IR£1.15. Subscription: 6 months: £26.00. 1 Year: £52.00. Binder: please send £3.95 per binder, or take advantage of our special offer in early issues. **EUROPE** – Issue Price: 90p. Subscription: 6 months air: £44.72. Surface: £36.14. 1 year air: £89.44. Surface: £72.28. Binder: £5.00. Airmail: £8.25. **MALTA** – Obtain binders from your newsagent or Miller (Malta) Ltd, MA Vassalli Street, Valetta, Malta. Price: £3.95. **MIDDLE EAST** – Issue Price: 90p. Subscription: 6 months air: £50.18. Surface: £36.14. 1 year air: £100.36. Surface: £72.28. Binder: £5.00. Airmail: £8.25. **AMERICAS/ASIA/AFRICA** – Issue Price: US/CANS1.95/90p. Subscription: 6 months air: £59.54. Surface: £36.14. 1 year air: £119.08. Surface: £72.28. Binder: £5.00. Airmail: £9.50. **SOUTH AFRICA** – Obtain binders from any branch of Central News Agency or Intermag, PO Box 57394, Springfield 2137. **SINGAPORE** – Issue Price: Sing $4.50. Obtain binders from MPH Distributors, 601 Sims Drive, 03-07-21, Singapore 1438. **AUSTRALASIA/FAR EAST** – Issue Price: 90p. Subscription: 6 months air: £64.22. Surface: £36.14. 1 year air £128.44. Surface: £72.28. Binder: £5.00. Airmail: £9.75. **AUSTRALIA** – Issue Price: AusS2.15. Obtain binders from First Post Pty Ltd, Locked Bag No. 1, Cremorne, NSW 2090. **NEW ZEALAND** – Issue Price: NZS2.65. Obtain binders from your newsagent or Gordon & Gotch (NZ) Ltd, PO Box 1595, Wellington.
ADDRESS FOR BINDERS AND BACK ISSUES – Orbis Publishing Limited, Orbis House, Bedfordbury, London WC2 4BT. Telephone 01-379 5211. Cheques/postal orders should be made payable to Orbis Publishing Limited. Binder prices include postage and packing and prices are in stirling. Back issues are sold at the cover price, and we do not charge carriage in the UK.
NOTE – Binders and back issues are obtainable subject to availability of stocks. Whilst every attempt is made to keep the price of the issues and binders constant, the publishers reserve the right to increase the stated prices at any time when circumstances dictate. Binders depicted in this publication are those produced for the UK and Australian markets only. Binders and Issues may be subject to import duty and/or local taxes, which are not included in the above prices unless stated.
ADDRESS FOR SUBSCRIPTIONS – Orbis Publishing Limited, Hurst Farm, Baydon Road, Lambourn Woodlands, Newbury Berks, RG16 7TW. Telephone: 0488-72666. All cheques/postal orders should be made payable to Orbis Publishing Limited, Postage and packaging is included in subscription rates, and prices are given in sterling.

COVER PHOTOGRAPHY BY JOE MUNROE © SCIENCE PHOTO LIBRARY

CAREERS FOR ENGINEERS

We look here at careers in hardware and software engineering, jobs that involve actually putting new systems together. We also focus on maintenance and technical support, the two career fields in the computer industry that are perhaps the easiest for young and inexperienced job seekers to enter.

A great deal of mystique surrounds the hardware engineer — the person who actually designs and builds computer systems — but exactly what he does can be broken down for easier definition.

The initial stage is that of specification — deciding which functions the new machine must carry out. This stage tends to be the realm of the more experienced engineer in conjunction with the marketing department. In the defence and military field (and in the UK more hardware engineers are employed in defence than in civil projects), the specification will be handed down by the Ministry of Defence.

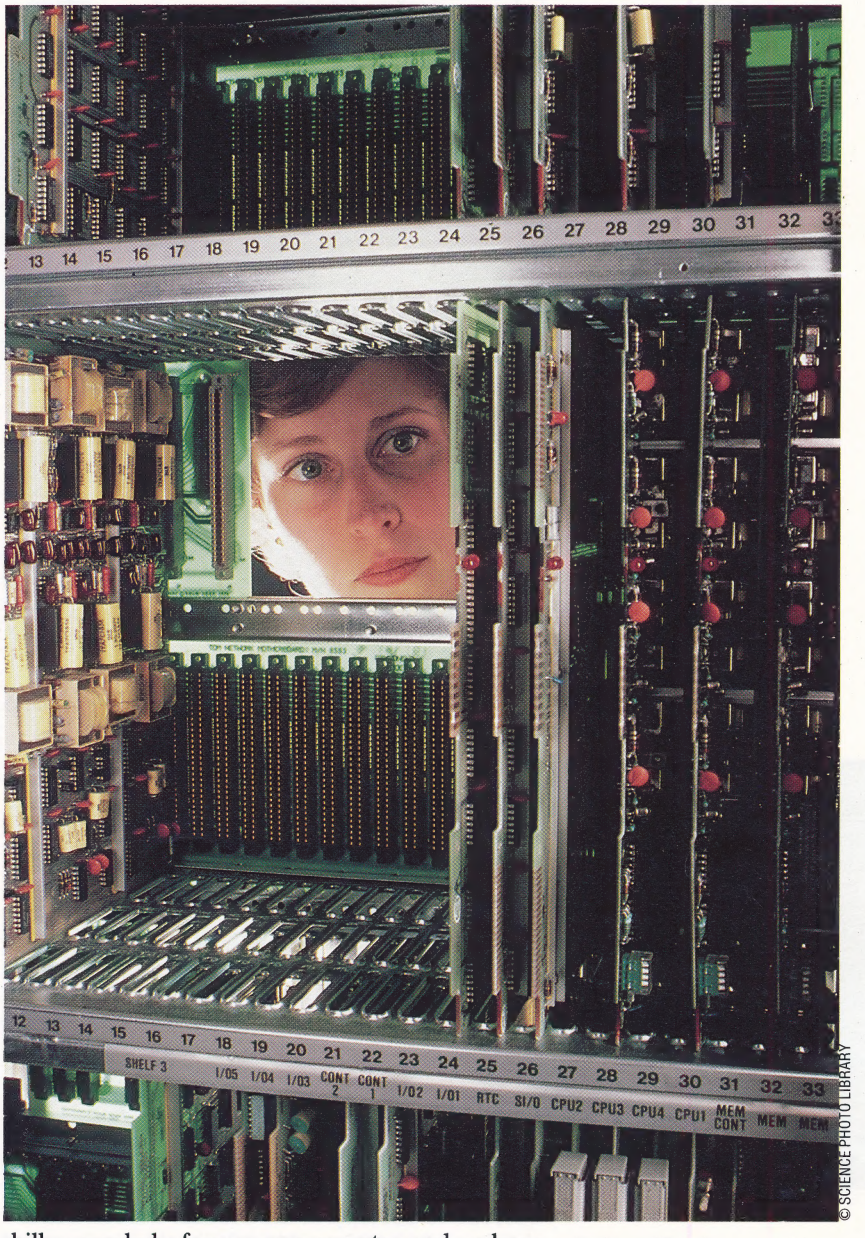
The next stage for the engineer, after receiving the specification, is to decide whether it can actually be carried out, and the time it will take. (Time is of the essence on all engineering projects, both military and civil). Having costed the project and assessed the time limitations, the engineer must then design the system. To decide on the components to be used and how they will be connected, engineers must be aware of the latest technological advances at component level and therefore need to be assiduous readers of the trade press. Following the initial design, several prototypes may need to be put together and tested before final production can begin.

DESIGNING PROTOTYPES

All engineering is inevitably a compromise that makes the best of limited resources. If you only have six months, you can't build an ideal system from scratch. If you have to build a product that is compatible with existing products, you won't be able to give it a revolutionary architecture. If a product will have to sell for a particular price, than there will be limits on the prices of the components used. It is the engineer's job, therefore, to design the best possible system within those limitations.

Hardware engineering is for the young. The 'midlife crisis' affecting workers in most industries, at 40 or 45, comes early. Many engineers feel that if they haven't branched out into management or sales by their late twenties, then they are failures. The 35-year-old engineer is often seen as a 'has been' who lacks the communication and personal skills needed for management, and whose knowledge of the technology is likely to be out of date.

Career entry to hardware engineering is at two levels. Many of the larger companies — particularly those in the defence field, such as Thorn EMI and British Aerospace — will take bright 16- or 18-year-olds with good A level or even O level grades and train them as apprentices. As with most reasonably well-paid professions, hardware engineering is becoming increasingly graduate-oriented. If you want to enter at graduate level, you'll need a degree in either physics or electronic engineering — hardware



© SCIENCE PHOTO LIBRARY

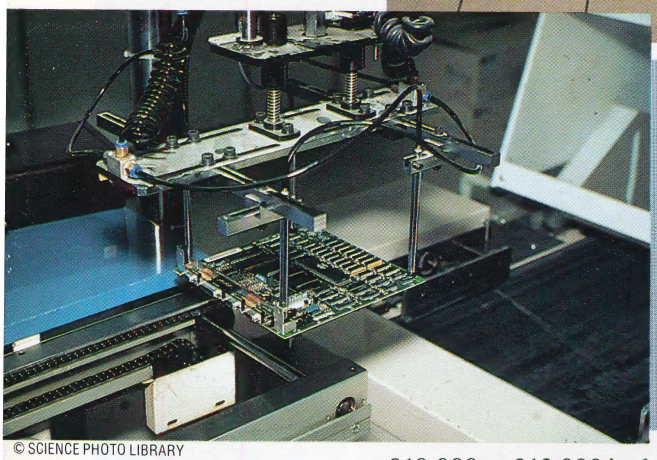
Designers' Boards
Most mini and mainframe computers are designed in modular fashion, making the technician's task rather easier. Different boards are dedicated to different functions, and test equipment will be available to monitor their performance

engineering is not a field open to maths, computer science, or other numerate graduates.

In general, defence pays less well than the commercial sector. Graduates or apprentices who have finished their training are likely to be taken on at salaries of between £8,000 and £10,000. Hardware engineering is unusual in that a year's experience is not enough to get a major salary hike — for the first two to three years, rises tend to be in the order of only 10 per cent. The experienced hardware engineer will earn in the region of £10,000 to £13,000 in the defence field and in the region of £13,000 to £16,000 in the commercial sector.

© SCIENCE PHOTO LIBRARY

Hi-Tech Help
Technicians are increasingly supplied with intelligent equipment to facilitate fault diagnosis and correction. Here a technician scans a Kray mainframe with an infra-red detector, looking for 'hot-spots' that could indicate a malfunction. Furthermore, many systems are now designed with in-built fault checking and easily replaceable modules, making the technician's role simpler and, in some cases, almost redundant



© SCIENCE PHOTO LIBRARY

Automated Design
The design engineer's role will include, at early stages of product development, the full consideration of the means of construction. A new product destined for mass sales must be designed to take full advantage of automated production techniques. The Apple Macintosh board shown here is being assembled by robot

Between the ages of 28 and 30, the engineer will tend to leave the field to go into management or sales, using his industry and technical skills. Salaries in sales can, of course, be particularly high, with salesmen earning up to £40,000. Project management is the first rung on the management ladder. After that, however, the engineer will become a manager and his technical skills will tend to become redundant.

There is a great deal of demand for hardware engineers, particularly overseas. In North America, salaries tend to be two to three times higher than those in the UK. High salaries are also offered by other West European countries — salaries in West Germany, for example, are up to twice those in the UK. Because English is the *lingua franca* of hardware engineering, lack of foreign languages needn't be a handicap.

Software engineering goes hand in hand with hardware engineering. Essentially, the software engineer's job is to provide the software environment — the operating system — that will work with the hardware. Often, the software engineer's role is to adapt an existing operating system that's brought in from a third party rather than to produce a completely new system.

Software engineers, due to shortages of trained staff, tend to be more highly paid than hardware engineers. Salaries start at around £10,000, rising up to £20,000. Entry is given almost exclusively to graduates, and unlike hardware engineering, numerate graduates are taken on and trained up.

The career crisis that hits hardware engineers in their late twenties also affects software engineers, though to a lesser extent. Again, the software engineer tends to go into sales, management or consultancy in his early thirties.

In general, hardware engineers can move into the software field while software engineers can't move over to hardware. The two job roles tend to be separated, though, with workers in each field collaborating on a project.

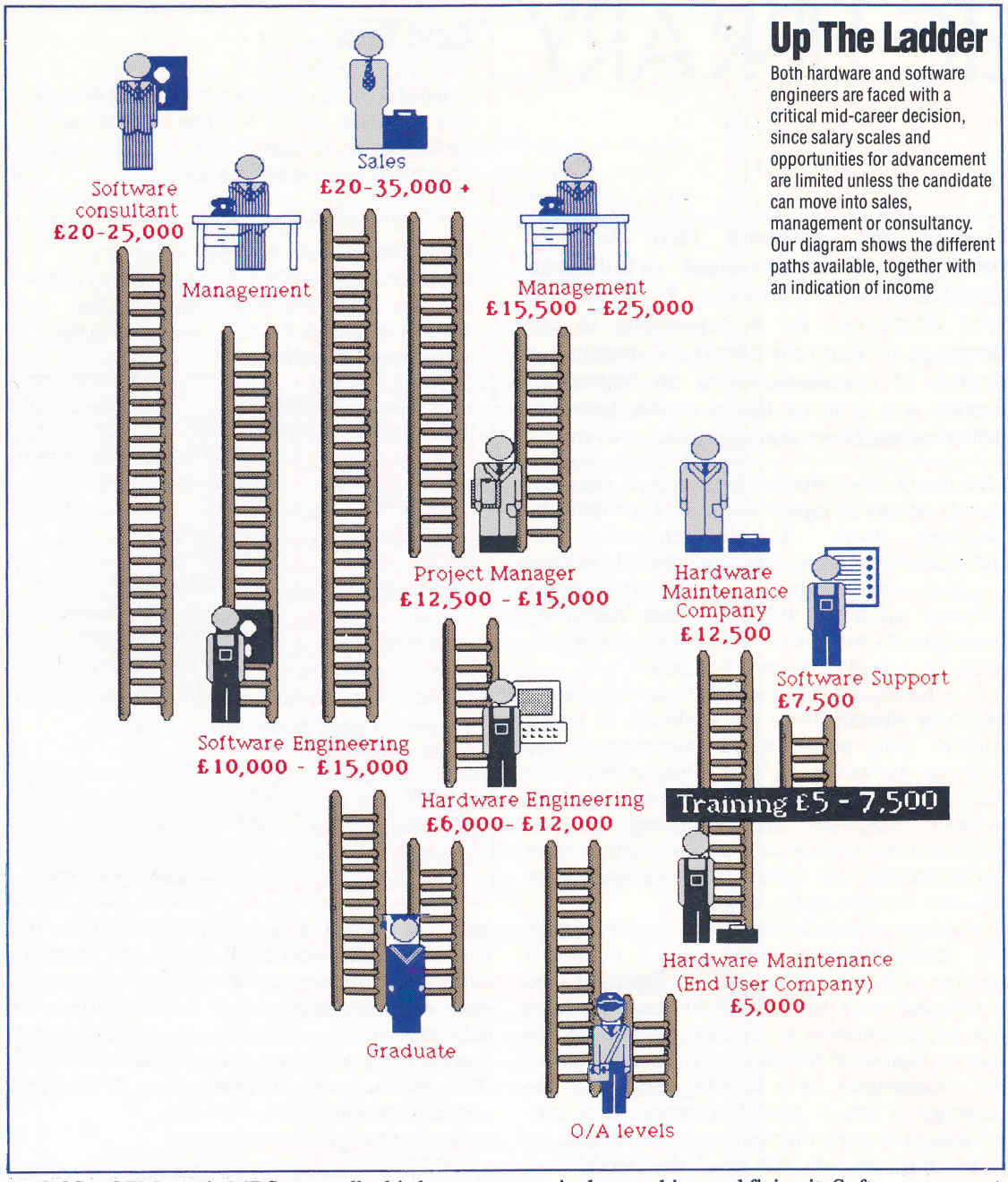
TECHNICAL SUPPORT

The technical support field provides a number of opportunities for those wishing to get into the computer industry who have some experience of home machines. As with engineering, there is a clear distinction between hardware and software maintenance and support. Of the two areas, hardware tends to be the poor relation, with lower salary levels than the software side.

Hardware support covers the installation of products at customer sites and the testing and diagnosis of hardware faults. Hardware support personnel will tend to work for either supplier or third party maintenance companies — those contracted to maintain and to 'trouble shoot' other companies' computers.

Almost all business micro dealers will have technical hardware support personnel and they will often take on keen young hobbyists. Starting salaries are quite low, with many jobs paying as little as £5,000. But salaries can rise to quite high levels; trained hardware support engineers, for example, could very well earn £15,000.

Within a small dealership, the support person will tend to get a lot of experience in software, in demonstrating products and in backing up sales staff. The young hardware support person in an IBM PC dealership or a minicomputer systems house can expect to be confronted with several career paths: the slightly higher paid software support business, going to work for one of the third party maintenance houses or making the move into sales.



GCS, CFM and MBS are all third party maintenance companies that employ large numbers of hardware support personnel. Typically, their engineers will be paid £15,000 plus a car, and get four weeks training a year in new systems and how to support them.

You should bear in mind a few points, however, before entering the hardware support field. Although it may seem unlikely at times, computers are in fact getting more reliable and are being designed to be easily supported. On some computers, the engineer simply has to replace one module with another, and self-diagnosis programs mean that less skill is required to find out what's wrong with a computer. Given that, hardware support personnel may well face technological redundancy in the near future.

Software support involves finding out why a particular piece of software isn't working on a particular machine and fixing it. Software support staff will also have to transfer programs from one machine to another and may get involved in specialised areas such as communications.

As with hardware support, relatively inexperienced people can get into the field. Essentially, there are three levels of entry — as a 16- to 17-year-old hobbyist willing to learn and receive a low salary for a couple of years; as an end-user with some experience of programming and using business computers; or as a graduate with a degree in a numerate subject (anything from geography to economics).

Apart from the first category, who will enter at a salary of around £4,500, starting salaries are in the region of £6,000 to £8,000, rising to £15,000 per year. A year's experience in software support is worth a lot, adding a couple of thousand pounds to your annual salary.

PUBLIC LIBRARY

Finding the optimum state between specifying all input/output details and specifying none is a problem C has dealt with very effectively. By incorporating in the language a 'standard library', C manages a degree of standardisation, an important feature if it is to be transportable between different machines and operating systems.

Like many other modern languages, C does not specify details of input/output (I/O) within the language itself. A comprehensive I/O specification for a language that's going to be used in a wide variety of applications over a number of different machines would be very restrictive, making many straightforward operations difficult, as in the case of COBOL and FORTRAN.

It is, for example, impossible to accept a single character directly from the keyboard in COBOL without using non-standard extensions to the language. However, if I/O is not specified at all, then it is difficult to make programs portable between machines and operating systems. Keyboard input in PASCAL is a good example of an operation that can behave quite differently on different versions of the language.

A degree of standardisation in C is provided by the standard library, stdio.h, which defines a number of high and low-level I/O functions. The details and exact provision of this may vary from one implementation to another, but it is best to stick to high-level functions. On the other hand, the programmer of a specific version of the language is free to provide functions that take advantage of particular aspects of a machine.

We have already discussed the 'workhorse' function for screen output (printf) and its related function for keyboard input (scanf). There are many other functions enabling more precise control of the keyboard and screen, as well as for I/O to disk files and other devices. C has simplified the idea of I/O to files and devices by treating them exactly the same. A file to C is a stream of bytes; input functions simply take a byte or group of bytes from an incoming stream, and output functions send out a stream. If the stream is connected to the disk, it is possible to move along the stream to a specified position, thus providing random access. When files are treated this way, the logical difference between a file and a device disappears.

Files must normally be opened before they are used, to specify the device or disk area where the byte stream is directed. There are, however, three files that are always open in any C program. These are: stdin, which is normally connected to the keyboard; stdout and stderr, which are normally connected to the screen. Many operating systems, such as Unix, allow input and output to be redirected so these assignments can be changed. The functions printf and scanf operate on stdout and stdin, but they are just special cases of the more general functions fprintf and fscanf.

So, for example, the statements:

```
printf(.....);
fprintf (stdout,.....);
```

are identical in operation. There are two other similar functions, sprintf and sscanf, which perform I/O to and from strings in main memory.

In C terms, stdout is actually a pointer to a data item of type FILE, which is normally defined by means of a macro (#define) in the stdio.h file. To open a file to disk or device, the function fopen (filename, filemode) is used. It takes a string for the filename that can refer to a device or file according to operating system conventions. The filemode can be "r" to read from the file, "w" to write to the file, or "a" to append to the file. If the file is opened with "a" or "w" and it does not exist, it will be created; a file that does exist and is opened with "w" will be overwritten. The file pointer — a long integer indicating the current position along the byte stream — will be positioned at the start of the file with "r" or "w". The value returned by the fopen

Acid Test

There are many useful functions and macros included in various libraries that we do not have the time or space to discuss. But there are a number of useful ones for character handling in the file ctype.h. They can be included with the line:

```
#include (ctype.h)
```

The first group provides a means of testing a character for particular attributes—returning a non-zero (true) value if the character tested has the attribute; otherwise, it is zero. They are normally implemented very efficiently

Name	True If
isalpha(c)	c is a letter
isupper(c)	c is upper case
islower(c)	c is lower case
isdigit(c)	c is a digit
isxdigit(c)	c is a hex digit
isspace(c)	c is a white space character
isalnum(c)	c is a letter or digit
ispunct(c)	c is a punctuation character
isprint(c)	c is a printable character
iscntrl(c)	c is a control character
isascii(c)	c is an ASCII code

The second group of functions provides three convenient conversions:

Name	Effect
toupper(c)	converts c to upper case
tolower(c)	converts c to lower case
toascii(c)	converts c to an ASCII code

Library Studies

The following are other I/O functions in C's standard library:

getc(pointer_to_file) Gets the next byte from the file or device that has been opened with "r". The value is returned as an int. A particular value EOF is returned if the end of the file is encountered. It may be implemented as a macro

getchar() This is equivalent to getc (stdin)

fgetc(pointer_to_file) Pushes the character c back onto the file, from which presumably it was read. It returns the int value of c. At least one character must have been read from the file before one can be pushed back, and it is not reliable if more than one character is pushed back at a time

putc(c,pointer_to_file) Outputs the character c to the output file, returning the int value of c. It may be a macro

putchar(c) Equivalent to putc(c, stdout)

fputc(c,pointer_to_file) Equivalent to putc but always implemented as a function

gets(s) Where s is a string (pointer to char), reads characters from stdin until a newline is encountered. The latter is not placed in the string, which will be properly terminated with a '\0'. The value of s is returned

fgets(s,n,pointer_to_file) Reads characters from the file into the string s until either n-1 characters have been read or a newline is encountered. The newline will be placed in s, which will be terminated with '\0'. The value of s is returned

puts (s) Outputs the string s to stdout appending a newline

fputs(s,pointer_to_file) Outputs the string s to the file with no newline

fseek (pointer_to_file, offset, place) Moves the file pointer along the byte stream to the offset or number of bytes from the specified place; place can be 0 for the beginning of the file, 1 for the current position or 2 for the end of the file. The offset should be of type long int

rewind(pointer_to_file) Equivalent to:
fseek(pointer_to_file),0L,0)

ftell (pointer_to_file) Returns the current offset (a long int) from the beginning of the file

unlink(filename) Removes the named file from the directory. It returns -1 if the file does not exist, and 0 otherwise

exit (status) Terminates a program returning the int value of status to the operating system or calling process. 0 is used for a normal termination

CAROLINE CLAYTON

function is a pointer to type FILE that can be used in other functions; it will be NULL if the file cannot be accessed for any reason.

There is a corresponding fclose (pointer_to_file) function that closes an open file. All files will automatically be closed when a program finishes, but this function may be necessary because of an operating system limit on the number of files that can be simultaneously open.

Character Assessment

```
/* this program will count the number of words,
and the number of characters in a file, whose
name is given on the command line */
#include (stdio.h)
#include (ctype.h)
main(argc,argv)
int argc;
char *argv;
   {
   int char_count = 0, word_count = 0, c,
   inword     = 0;
   FILE *in_file, *fopen();
/* note the file name and the fopen function
declared as pointers to type FILE */
/* check for right number of arguments*/
   If (argc != 2)
   {
      fprintf(stderr,"\nusage is %s filename\n",
      *argv);
/*remember that the first entry in array argv is the
actual program name*/
      exit (1);
   }
/*open the file and check if it exists, ++argv
points to the name of the file*/
   if ((in_file = fopen(*++argv,"r")) == NULL)
{
      fprintf(stderr,"\ncannot open %s\n",
      *argv);
/* remember argv is now pointing at the name of
the file*/
   exit(1);
   }
   while ((c = getc(in_file))     != EOF)
   {
      ++char_count;
      if(inword)
      {
         if (isalnum(c))
         ;/*an empty statement */
         else
         {
            inword = 0;
            ++word_count;
         }
      }
      else
         if (isalnum(c))
            inword =1;
   }
   if (inword)
      ++word_count;
   printf("\nnumber of chars = %d",char_
   count);
   printf(\nnumber of words = %d\n",word_
   count);
   fclose (in_file);
}
```

KEEP YOUR OPTIONS OPEN

Unix has an enormous variety of tools and utilities that can be used singly or in combination, using the techniques of redirection and piping discussed in the previous instalment. Here we turn our attention to those tools concerned with file management and text preparation.

The general form of a Unix command is:

command_name options arguments

Each portion of the command must be separated from the next by at least one space. Arguments to a command are usually file or directory names; if the argument is omitted, the standardinput and standardoutput files (the keyboard and screen) are used for input and output by default.

Options take the form of single letters, preceded by a hyphen, and are used in conjunction with the command to perform a range of tasks. Where options do not require other information, such as a file name, more than one option may follow the hyphen. For example, the directory listing command ls has a number of options (see page 1808), including l, which produces a full listing, and a, which lists file entries. Arguments to ls can be either a file specification or a directory, and the current directory will be taken by default if no argument is included.

To list the contents of the /usr directory, with both options selected, the command could be one of the following:

```
ls -l -a /usr

ls -la /usr
```

Incorrectly issued commands result in an error message, which tells the user that the command is unrecognised, or gives the correct usage if possible. Note that two or more commands can be written on the same line by separating them with a semicolon.

FILE MANAGEMENT COMMANDS

The command wc counts the number of characters, words and lines in a text file. The options are:

- **l** to count lines only
- **w** to count words only
- **c** to count characters only
- **p** to count pages (one page is 66 lines)

The arguments may be one or more file names. If more than one file name is given, each file is counted and a total is given for all the files specified. If no file is given as an argument, ws assumes that the input comes from the keyboard.

On standardoutput (normally the screen), head displays the first few lines of a file. The only option is one that determines the number of lines to be displayed. For example, -15 displays the first 15 lines. The arguments must include one or more file names.

The tail command gives the last few lines of a file. The options are:

- **+n** an exception to the rule that options always start with -; the remainder of the file is displayed from a point 10 lines from the start
- **-n** displays the final n lines in the file; the default is 10
- **l** counts in lines (the default)
- **b** changes unit to blocks of disk storage
- **c** changes unit to characters
- **r** displays the file in reverse order

The sort command sorts a file into key order, or sorts and merges several files The options are:

- **b** to ignore leading spaces
- **d** dictionary order, using only letters, digits and blanks
- **f** make case-insensitive
- **n** sort numbers by arithmetic value rather than by digits
- **o** directs the output to a file rather than standardoutput
- **r** sort in reverse order

The arguments are one or more file names; if only one file name is given, the contents of that file are sorted. Specifying multiple file names causes the files to be sorted and merged together.

The cmp command compares the contents of two files to determine any differences between them. When operating in its default mode, the command returns the byte and line number when the first difference is detected. The only option, l, reports on all the differences throughout the two files. If one file name is omitted, standardinput is assumed.

The comm command looks at two files that should be sorted into ASCII order and displays three columns: those lines only in the first file; those only in the second file; and those lines common to both. The options are simply 1, 2 and 3 to omit one of the three columns. Two file names must be specified as arguments.

The diff command also finds the differences between two files. It indicates what changes must be made to the first file to make it identical to the second, using a for append, c for change, d for delete, < for a line from the first file, and > for a line from the second file. The options are:

- **b** to ignore trailing blank spaces and to equate strings of blanks regardless of length
- **e** to produce output as commands for the editor
- **r** is only used with directories, and allows diff to apply itself recursively to any subdirectories

The arguments may be either a pair of file names or directory names. If the arguments specify two directories, diff lists all files unique to each directory, and then lists in the third column those files common to both.

The uniq command compares adjacent lines within a text file and removes duplicated entries. The options are:

- **u** to display only those lines with no duplicates
- **d** to display just one copy of duplicated lines and no unique lines
- **c** to accompany each line of output by the number of times it appears within the file

The arguments may be one or two file names. If two files are specified, the output goes to the second named file, rather than to the screen (standardoutput).

The lpr command sends one or more files to the system printer. Because the output from multi-tasked jobs cannot be made directly to the printer, they are placed in a queue. Unix scans the queue continually and prints the file output at the head of the queue, which is known as 'spooling' (see page 1660). The arguments are one or more file names. There is no standard set of options since printing facilities vary widely between installations.

The lpq command displays details of the current state of the print queue, which enables you to check whether a particular file has been printed, or how long it is likely to remain in the queue awaiting printing. Each printing job in the queue is assigned a number and there are no arguments or standard options.

The lprm command allows the removal of a file from the print queue before it is printed. The argument can either be a file name, the print job number obtained using lpq, or a login name, in which case all files belonging to that owner will be removed.

The command pr displays the contents of the file on standardoutput, formatted for printing. The text is organised into pages, with five lines of bottom margin and a heading, consisting of the date, file name and page number followed by two blank lines. Commonly, the output would be piped into lpr. The options are:

- **n** to organise the text into a given number of columns
- **m** to display two or more files side by side
- **t** to suppress the heading and bottom margin

The arguments are one or more file names. What follows is an example of the power and efficiency of Unix as it deals with the manipulation of three text files.

File Manipulation

```
%cat file1
The cat sat on the mat.
Mary had a little lamb.
The quick brown fox jumps over the lazy dog.
The owl and the pussy cat went to sea.

%cat file2
The cat sat on the dog.
Mary had a little lamb.
The quick brown fox jumps over the lazy dog.
The owl and the pussy cat went to sea.

%cat file3
The cat sat on the mat.
Mary had a little lamb.
The quick brown fox jumps over the lazy dog.
This file has an extra line.
The owl and the pussy cat went to sea.

%wc file1       (count the number of lines, words and characters)
      4      29      132 file1

%wc -w file1    (count only words)
      29 file1

%head -2 file1  (display first two lines only)
The cat sat on the mat.
Mary had a little lamb.

%tail +2 file1  (display from the second line)
Mary had a little lamb.
The quick brown fox jumps over the lazy dog.
The owl and the pussy cat went to sea.

%tail -2 file1  (display last two lines)
The quick brown fox jumps over the lazy dog.
The owl and the pussy cat went to sea.

%sort file1     (sort file into order)
Mary had a little lamb.
The cat sat on the mat.
The owl and the pussy cat went to sea.
The quick brown fox jumps over the lazy dog.

%pr file3       (format a file for printing)
The cat sat on the mat.
Mary had a little lamb.
The quick brown fox jumps over the lazy dog.
This file has an extra line.
The owl and the pussy cat went to sea.

%sort file1 > file4  (create sorted versions using redirection)
%sort file2 > file5

%cmp file1 file2     (compare a pair of files)
file1 file2 differ char 20, line 1

%comm file4 file5
                                Mary had a little lamb.
                        The cat sat on the dog.
The cat sat on the mat.
                                The owl and the pussy cat went to sea.
                                The quick brown fox jumps over the lazy dog
(lines in file4/lines in file5/lines in both files)

%diff file1 file2  (display differences between files)
1c1                ('c' implies 'make a change')
< The cat sat on the mat.
---
> The cat sat on the dog.

%sort file2 file3 > file7     (merge two files)
%cat file7
Mary had a little lamb.
Mary had a little lamb.
The cat sat on the dog.
The cat sat on the mat.
The owl and the pussy cat went to sea.
The owl and the pussy cat went to sea.
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
This file has an extra line.

%uniq file7         (display file ignoring repeats)
Mary had a little lamb.
The cat sat on the dog.
The cat sat on the mat.
The owl and the pussy cat went to sea.
The quick brown fox jumps over the lazy dog.
The file has an extra line.
```

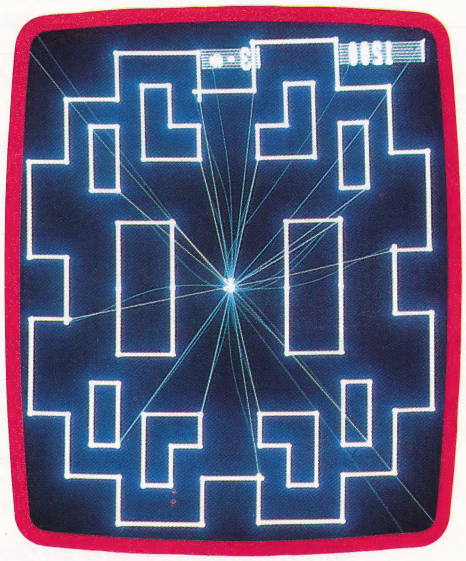
VECTOR

A *vector* is a one-dimensional array. Since a computer's memory is essentially linear, it can therefore be considered a vector. The term is also used in relation to interrupt systems. Interrupt vectors are often used in computer design to manage the many devices and peripherals that may require the processor's attention.

When a device generates an interrupt, it supplies the processor with an address in memory that contains another address, which is the interrupt vector. This points to the location of the interrupt handler routine. If the interrupt vector is held in RAM, it is possible to alter its value so that it points to a user routine.

VECTOR GRAPHICS

A method of displaying graphics on a specially designed monitor, *vector graphics* involves 'drawing' the image on-screen with the electron beam in a cathode-ray tube. This is in contrast to the standard 'raster scanning' technique (see page 1440), whereby the picture is constructed as the electron beam passes over the entire screen, a line at a time.



LIZ HEANEY

Points Of View
Vector graphics are characterised by the monochrome lines drawn by the electron beam to a series of points. On the example screen shown here, the brilliance has been deliberately over-emphasised in order to show the beam and the points to which each of the lines is plotted

Vector graphics are displayed as the computer plots a series of points on the screen. The graphics can then be drawn by sweeping the electron beam from point to point, thereby creating the image. The technique enjoyed a brief heyday in the late 1970s in a number of arcade machines — in particular, Battlezone and Asteroids — in which the simple outlines produced by vector graphics made a welcome change from the block graphics of Space Invaders. The process quickly fell out of favour with the arrival of true colour monitors.

VERIFICATION

The process of checking whether information has been transferred correctly from one device to another is termed *verification.* On home micros, the process of verification is most often used when programs and data have been transferred to cassette or disk.

In order to make a verification, the computer will read back the written data and compare it with the copy of the information that it holds in memory, or will compare checksums.

VIDEOTEX

A *videotex* system allows you to transmit data to a VDU screen via cable or radio waves. The essential difference between a videotex system and teletext (see page 1749) is that the former is 'interactive' whereas teletext 'passively' receives information.

In order for a user to communicate with a videotex system, the user's terminal must be equipped with a VDU screen (a monitor or television), a keyboard and a method of coding and decoding the signals. By far the most widely used form of transmission and reception is via a modem and the telephone network.

Because many videotex systems use high resolution graphics and images, a high-quality modem is required using a 1200 baud full duplex system. But for some videotex systems, a lower baud rate will suffice, entailing lower costs all round.

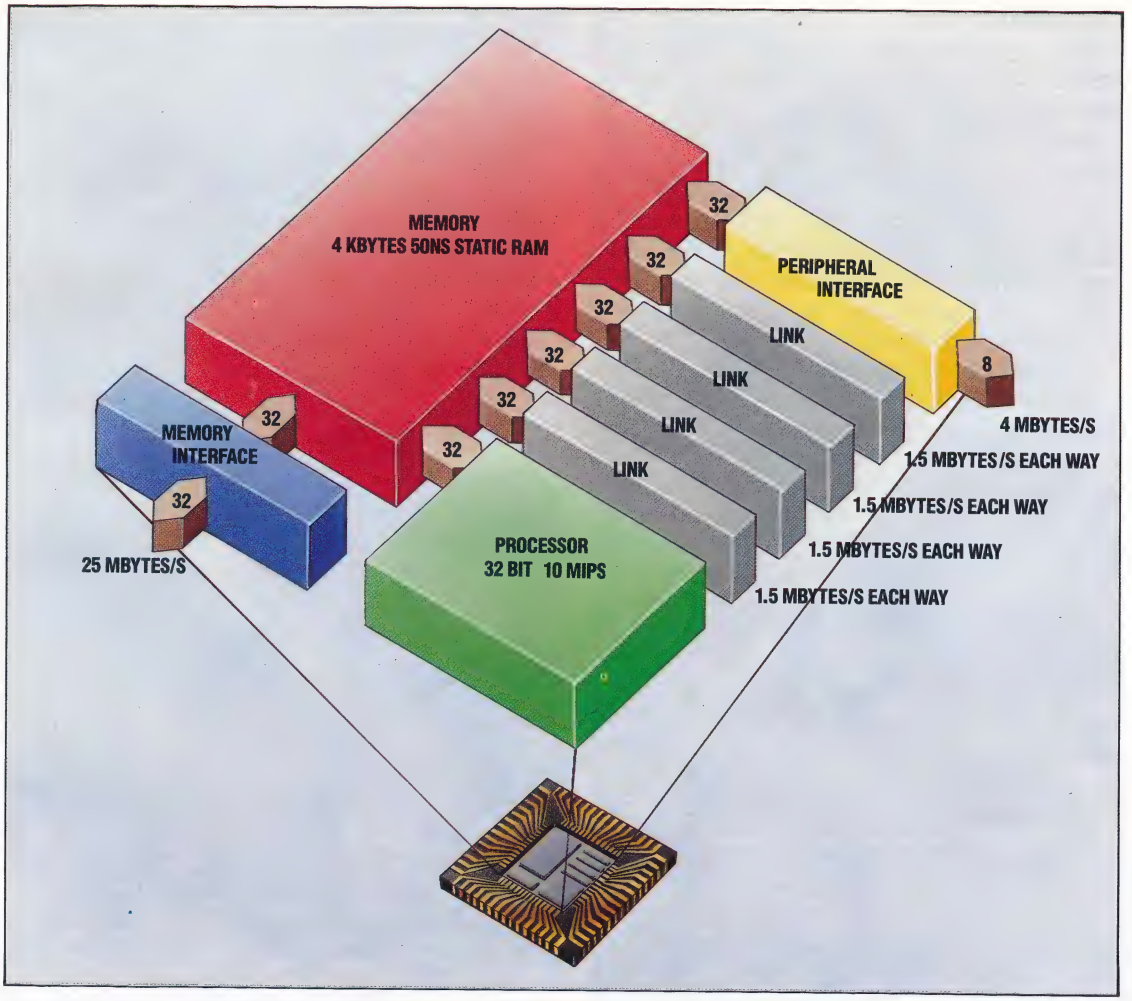
Although the prospect of home shopping and banking seems imminent, the great majority of videotex users are businesses, such as travel agents and financial institutions, which have access to large computer systems and are able to react rapidly to changing conditions.

VIRTUAL MEMORY

Traditionally, *virtual memory* referred to a technique whereby the full addressing power of the processor could be utilised without having to use large quantities of expensive silicon-based memory. When the processor attempts to access data not present in RAM, the system will react by switching over to another area of memory actually held on a cheaper magnetic storage system, such as tape or floppy disk. The information from the magnetic medium will then be copied to an area of RAM where it could be accessed by the processor.

An example of this kind of memory management can be seen in a program such as WordStar, a large quantity of which is held permanently on disk. The software is arranged so that parts of the program are continually being swapped in and out of main memory, giving the impression that the software occupies more memory than is actually available.

More recently, the reason for using virtual memory management has been reversed. The price of RAM chips has fallen dramatically over the past few years, and as a consequence, microcomputers are being equipped with enormous quantities of main memory. Operating systems, such as MS-DOS, are incapable of addressing these amounts of memory. The result is that computers like the IBM PC/AT configure the memory beyond the 640Kbytes addressable by MS-DOS as a RAM disk. Virtual memory management techniques are then used to access this memory.



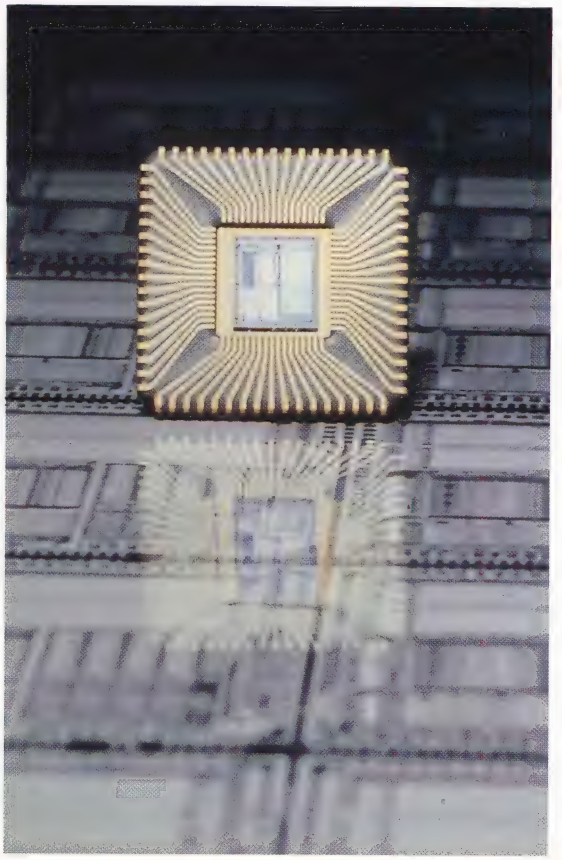
KEVIN JONES

Transcendental Computation
The Transputer, a new development from Inmos, is a computer on a chip, equipped with a 32-bit processor, RAM and four serial I/O links. The Transputer is regarded as a component to be used within the architecture of much larger parallel computers. The serial links enable a number of Transputers to be connected together in networks, the topology of which will be determined by the task for which the system is designed

CHIPS WITH EVERYTHING

The impact transistors had in the 1950s on the development of electronics is about to be equalled, if not surpassed, by the introduction of the Inmos Transputer. We look here at the technology that will generate processing speeds hitherto unthinkable.

The invention of the transistor in the 1950s opened up the way to mass-produced computers. Today's microprocessors are built up from hundreds of thousands of identical transistor switches made on the same slice of silicon. The Inmos Transputer is a revolutionary new microprocessor intended to be used in a similar way to the transistor — as a component or building block for larger systems (the name suggests a hybrid of *trans*istor and com*puter*). Parallel computers can be built by combining large numbers of Transputers, which are treated as 'programmable components' rather than as single

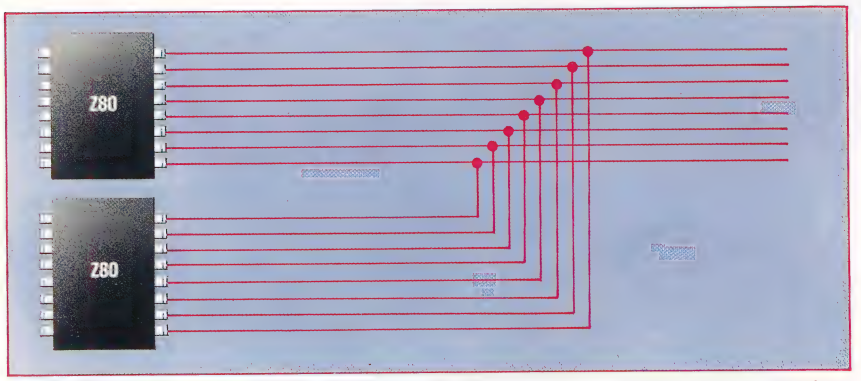


omnipotent CPUs.

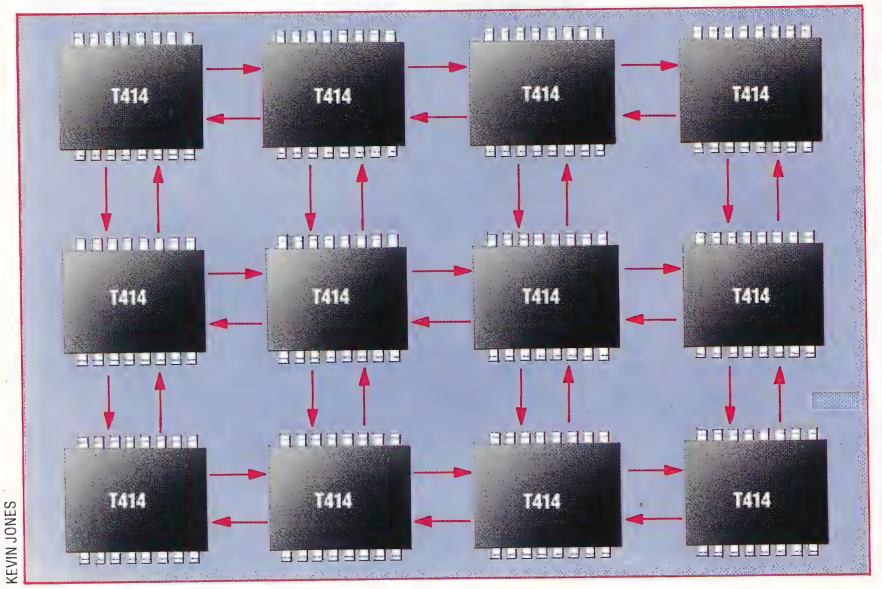
Such a combination is possible because the Transputer includes on a single chip all the components of a computer system. Each Transputer has a CPU, memory and serial communications links all on the same piece of silicon. This means that a single Transputer can execute a program on its own, with no need for any outside resources (except electrical power and a clock). Furthermore, Transputers can talk to each other much more easily than can traditional CPUs, via their serial links.

A traditional microprocessor, such as the Z80, is the 'boss' in any system in which it is used. Communication with the outside world is done over a parallel bus, access to which is controlled by the CPU itself. It's difficult to make several Z80s talk to each other for two reasons. First, it's physically difficult to design a circuit board on which many chips share a parallel bus; eight wires need to be joined every time a connection is made. This problem gets worse rather than better with the more modern 16- and 32-bit chips.

Secondly, each Z80 wants to control the bus, so software must be designed to make sure that two processors do not try to talk at the same time. The Z80 cannot do any other job while it is communicating over the bus.



The Transputer, by contrast, needs only two wires to join each pair of chips to talk to each other.



KEVIN JONES

The Von Neumann Architecture

All the computers that we use today, from the humblest home machine to the largest mainframe, are based on the principles put forward by John von Neumann in 1945. A von Neumann computer consists of a central processing unit (CPU), connected to some memory that holds both the program instructions that specify what is to be done and the data to which it will be done. The instructions are fetched one after the other, or sequentially, and executed by the CPU.

This so-called von Neumann architecture was responsible for making general-purpose computers a practical proposition. The key idea, that numbers in memory can represent either a program or data, liberated computer designers who had previously built machines dedicated to particular jobs, like code-cracking or machinery control.

The problem lies in the sequential way that a von Neumann machine fetches and executes its instructions. However fast we make the CPU run, it cannot work any faster than it can fetch its instructions and data from the memory, and there are physical limits on how fast this can be done. In this way the rate of communication between a single sequential processor and its memory has become a bottleneck.

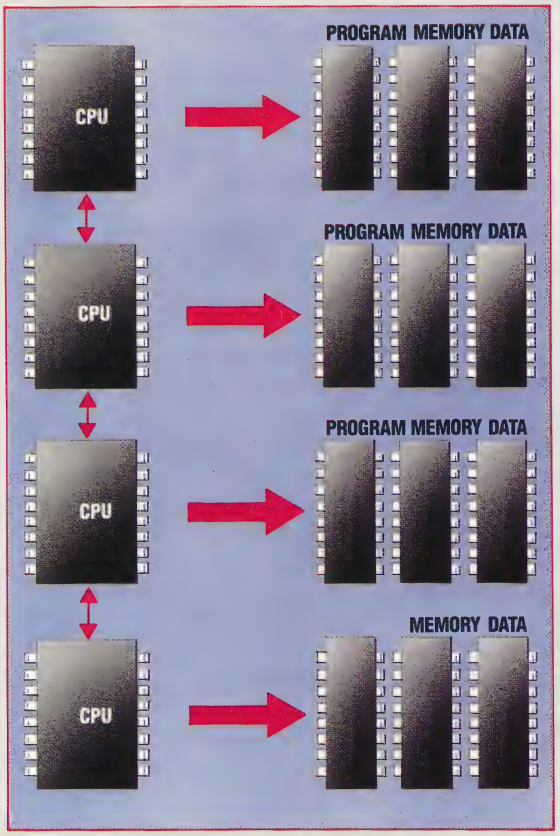
The bottleneck can be avoided by designing computers with more than one CPU, and having them all working at the same time, which is known as parallel processing (see page 1228). Even though the speed of communication with memory is limited, 100 CPUs talking to 100 memories can do 100 times as much work in a given time as a single processor can. Parallel computers gain even more speed because each CPU needs to talk to less memory, and this speeds up the maximum communication rate, as shown in the diagram.

Parallel processing presents difficulties that have kept it from being widely adopted. To execute a single program on a number of CPUs, each processor must be given a part of the job to do, and must do its part without clashing or competing with its neighbours. For example, if we wished to use two

Since each Transputer has four built-in serial links, it can talk to four other chips. This allows arrays of two or three dimensions of Transputers to be built with a minimum of wiring.

Most importantly, the Transputer links are designed with communication in mind, and no special software is required to prevent clashes. A single Transputer is capable of doing many things at once; in particular, it can receive messages on all four links simultaneously without having to stop whatever else it is doing at the time.

These design features allow a single program to be broken down into parts and run on a number of Transputers. Each chip can execute its own part of the program in its own memory, and then inform its neighbours of the results by sending messages over the links. If you need to execute a particular program faster, then you need only add more



processors to calculate the value of the expression :

(3 + 4) * (9 – 7 + 3)

each could take one of the bracketed expressions and calculate it, and then one of them could do the final multiplication.

It's essential to agree in advance which CPU will perform the final step. Also, CPU1 must not try to perform the last step until CPU2 has finished its part of the calculation. It's necessary for the CPUs to talk to each other and pass each other partial results and signals. These sorts of problems are not easily dealt with by traditional programming languages, which assume that everything will happen sequentially. Past attempts at parallel computers have often proved disappointing in performance

Big Country

Technology: The T414 is fabricated in 1.5 micron CMOS (complementary metal oxide/silicon) technology. This makes it one of the densest and most complex ever made

Processor: A 32-bit microprocessor that can execute 10 million instructions per second (approximately 10 times faster than the Motorola 68000, for example). It uses only 70 instructions that directly support the OCCAM language

On-Chip Memory: Static RAM with a 50 nanosecond access time. This is used to execute small OCCAM processes; it replaces the registers of a conventional microprocessor

Off-Chip Memory: Up to four gigabytes of external memory can be addressed, allowing the Transputer to be used as a stand-alone CPU chip in personal computers. The processor sees no difference between on-chip and off-chip memory except for access speed

Serial Links: Each link is capable of transferring 10 megabits per second in both directions; all four links can operate simultaneously giving a potential total data rate of 80 Mbits per second. Communication is asynchronous with a hardware handshake, so the Transputers at each end of a link do not need to share the same clock signal

Concurrency: A single T414 can run multiple processes at the same time, controlled by a built-in process scheduler. All the parts of the chip run concurrently with one another, so that the processor part can perform calculations or access RAM while messages are being sent on the links

Other Products: The T414 is the first of a family of compatible Transputer products in which various special-purpose functions will replace some of the memory or links. Two already designed are the G213 Graphics Processor and the M212 Disk Controller

The Company: Inmos was set up with the assistance of the British government in 1978 and is now part of the Thorn EMI group. The design and development of the Transputer and OCCAM were largely performed in Bristol, and the Transputer is manufactured in Newport, Wales

Transputers.

The key that enables the Transputer to be used in this way lies in the language used to program it. The Transputer does not have an assembly language like traditional microprocessors, but executes a specially designed language called OCCAM, which is tailored to writing parallel programs. An OCCAM program is divided into parts called 'processes', which are rather like subroutines or procedures in BASIC or PASCAL. The big difference is that processes can be executed at the same time as well as in a serial fashion.

In addition to storing values in variables, like a 'conventional' language, OCCAM can communicate values over channels. So different processes executing at the same time can communicate results to each other over a channel. Moreover, OCCAM channels are self-synchronising so that it is impossible for communication to occur until both partners are ready. OCCAM also has the vital property of not discerning whether the processes that make up a program are executed on the same Transputer or on different Transputers; a channel may be just a 'mailbox' in memory in the first case, or a piece of wire in the latter.

The Transputer offers a radical new approach to the design of high-performance computer systems. It blurs the distinction between software and hardware design, for anything that can be described by an OCCAM process can be implemented by a chip. Achieving performance goals becomes a matter of choosing the right size and topology (or 'shape') of Transputer network to fit the problem. The future might offer rings, 'doughnuts', cubes and other configurations.

HOPE IN TIMES OF CRISIS

Functional programming, which alleviates the problems associated with the enormous programs needed to run today's computers, is particularly suited to systems based on parallel processing. Here we look at the principles behind HOPE, a functional language developed to cope with the present 'software crisis'.

Many computer scientists are experimenting with *functional* programming languages as a way out of the 'software crisis'. Programs written in functional languages have the property that any part of a program can be understood without referring to the rest of the program; they do not depend upon the history or order of execution of their parts.

This 'transparency' is achieved by abolishing the use of assignment to variables. Some functional languages ban variables altogether and employ 'functions', which return values for immediate use. The principle can be illustrated even in BASIC by these two programs:

```
10 X = 57
20 A = SIN (X)
30 B = LOG (A)
40 C = SQRT (B)
50 PRINT C
```

```
10 PRINT SQRT (LOG(SIN(57)))
```

In the second program, no variables are used but the result is the same as in the first. The program consists solely of the application of functions to the results of other functions.

The only well-known functional language is the original 'pure' form of LISP (see our series beginning on page 1376), but most modern dialects have added non-functional features such as assignment with SETQ and loops to speed execution on conventional computers.

A good example of a modern purely functional language is HOPE, which was invented at Edinburgh University. HOPE programs are written by defining functions, as in LISP. Each function consists of a series of equations, which tell the function what value to return for every possible form of its arguments. Variables are allowed in HOPE programs but they cannot have their values changed by assignment; the only way to give a value to a variable is by using it as a 'pattern' that matches the arguments of the function.

For example, a function to calculate the square of a number might be written as:

```
      dec square : num—> num;
---   square (x) <= x * x ;
```

where dec stands for 'declare' and starts the definition; num—> num says that the function square takes one number as its argument, and returns one as its value (values in HOPE, as in PASCAL, have a 'type'). The equation (introduced by ---) simply says that the square of any number is that number multiplied by itself; the symbol <= means 'is defined as' or 'could be replaced by'. The pattern x on the left-hand side matches any number given as an argument. The right-hand side of an equation must be an expression, and may use only variables from the pattern.

We use the function by entering, say, square (4) ; and HOPE replies with 16 : num, giving the type as well as the value of the result. Of course, this function can be used in defining further functions.

A slightly more complex example is a HOPE function to calculate the factorial of a number:

```
      dec fact : num—> num ;
---   fact (0)          <= 1 ;
---   fact (succ(n))    <= (succ(n) * fact(n)) ;
```

The two equations define the function's value for all possible cases (type num represents positive integers so the negative case doesn't arise). In the case of its argument being 0, then it returns the value 1. In any other case the factorial of 'one-more-than-n' is 'one-more-than-n' times the factorial of n. This is a recursive definition because fact is defined in terms of itself; functional languages use recursion in place of looping.

The order of the equations is irrelevant, and the following works just as well:

```
      dec fact : num—> num ;
---   fact (succ(n)) <= (succ(n) * fact (n)) ;
---   fact (0) <= 1;
```

The function succ (for 'successor'), which returns a number one more than its argument, is built into HOPE and is called a *constructor* function. Every data type in HOPE is built by its own constructor function. When we write a constant like 3, we are in fact evaluating a function called 3 whose value is 3, but which is really a shorthand version of the expression succ(succ(succ(0))).

HOPE uses *lists* instead of arrays to represent multiple objects of the same type. Lists are written in brackets, so [1, 2, 3, 4] is a list of four numbers. A list can be matched by a pattern — such as x:: y, where x matches the first element of the list, and y matches all the rest. Using the [1, 2, 3, 4] list, x would be 1 and y the list [2, 3, 4]. The : : symbol, pronounced 'cons', is the constructor function for lists.

Text strings are represented as lists of characters and can be alternatively written in quotes, so "fred"

means the same as ['f','r', 'e', 'd']. A function to count the number of letters in a word could be:

```
        dec lettercount : list char—> num;
---- lettercount (nil)  <= 0 ;
---- lettercount (x :: y)<= lettercount (y) + 1;
```

where nil means an empty list. It is used like this:

```
letter count ("aardvaark");
9 : num
```

Types in HOPE are much more flexible than in PASCAL, and it's possible to write functions that can work on any type. For example:

```
typevar : alpha

        dec listcount : list (alpha) —> num;
---- listcount (nil)    <= 0;
---- listcount (x :: y)  <= listcount (y) + 1;
```

will count the elements of a list of any type and could be used in place of lettercount.

Programmers may define their own data types, of any complexity, and these can be passed as arguments or returned as values from functions. It is even possible to pass functions as arguments and return them as results, which allows extremely powerful programs to be written.

The Software Crisis

The 'software crisis' is a dramatic way of describing the fact that as computers get faster and cheaper, the cost of writing software for them continues to grow and, even worse, so does the cost of maintaining existing software. In fact, the reliability of large software systems has hardly improved since the early days of computing.

The roots of the software crisis lie in the nature of conventional programming languages. Although modern structured languages like PASCAL have brought about some improvement, it is still difficult to understand what a program does simply by reading its source code. With assembly language and unstructured languages like BASIC or FORTRAN, the problem is even worse. Even the author of a program may have problems reading his work later, while other people who have to maintain such programs face an enormous task. For large software systems like those used in space exploration or defence installations, it is doubtful if anyone alive understands the program in its entirety.

Much of the problem in reading programs stems from the fact that they are 'history-dependent'. Unlike a mathematical formula, a computer program text doesn't always convey all the information required to understand what it does. The value of a variable in a conventional program depends upon the previous 'history' of execution of the program. Take for example this program in BBC BASIC:

```
10  FLAG% = 0
15  REM
20  DEF FNx (A%)
30     FLAG% = 1
40     = 2 * A%
45  REM
50  DEF FNy (A%)
60     LOCAL B%
70     IF FLAG% THEN B% = 3 ELSE B% = 4
80     = B% * A%
85  REM
90  PRINT FNy (2)+FNx(1)
100 PRINT FNy (2)+FNx(1)
```

The two PRINT statements will print the values 8 and 10 respectively; the value of FNy(2)+FNx(1) is not the same in both cases. Changing the order of the terms also alters the value of the expression, so that FNy(2)+FNx(1) is not the same as FNx(1)+FNy(2). To understand most programs, we have to mentally 'execute' them using a pencil and paper.

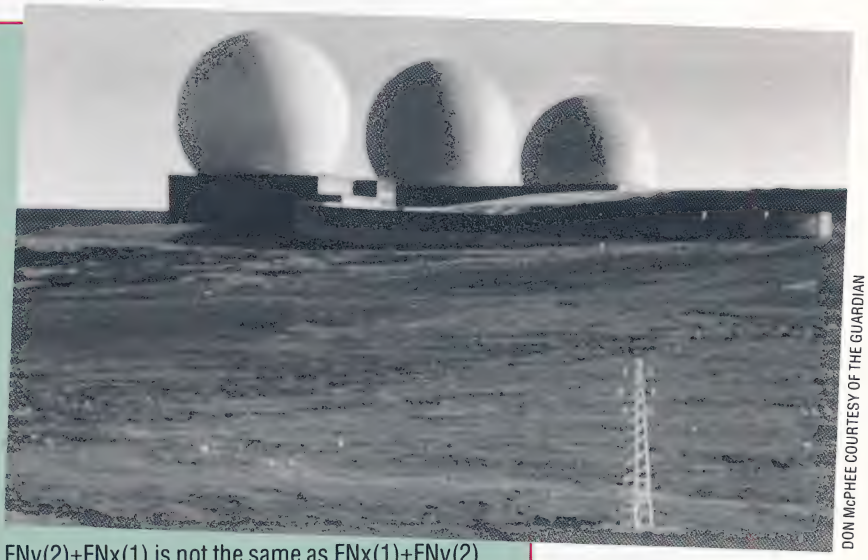
In contrast, mathematical statements are independent of any history. We know that the expressions:

```
(3+4) * (6–2) = (4+3) * (6–2) = 7 * 4 = 28
```

all have the same value because mathematical 'laws' tell us that 3 + 4 is the same as 4 + 3, that 7 can be legally substituted for either expression, and that the value of an expression is the value of its components. Programs written in conventional computer languages do not possess any of these properties.

If programs could be made to possess these mathematical properties, then several immense benefits would result and a solution to the software crisis would be in sight.

- Programs would become readable. The text of the program would express its meaning without any need to trace the history of its execution.
- Programs could be proved (like theorems) to be correct for all possible inputs, rather than merely being tested for a few of the possible inputs. This proof might be automated and performed by the computer.
- Inefficient but correct programs could be transformed into more efficient ones using laws like those for mathematical expressions. Again, this process might be automated



DON McPHEE COURTESY OF THE GUARDIAN

A Round Of Golf
Producing software for large computer-based systems, such as the Fylingdale's early warning system in North Yorkshire, is a major problem. Attempting to connect and rationalise the work of many programmers into a single large piece of code using traditional programming techniques is very difficult, and later modification can become an enormous headache. It is hoped that functional languages will provide the final solution to this problem

Where There's Life
The functional language HOPE was developed in the Computer Science Department of Edinburgh University By R.M. Burstall and D.B. MacQueen, with D.T. Sannella from Bell Labs in the US. The language is purported to be named after Hope Square in Edinburgh, where the Computer Science Department is situated

Along The Same Lines

One of the great attractions of the functional style of programming is that it lends itself to parallel execution. This is because the parts of a functional program are independent of each other, thanks to the lack of variables.

In conventional languages, the use of shared variables by different parts of a program poses enormous problems for a parallel computer. Let's suppose that we have invented a form of parallel BASIC and that these two programs are running simultaneously:

```
10   A = 0                    10   B = 0
20   FOR X = 1 TO K           .
.                             .
.                             .
100  NEXT X                   .
2000 A = B + 56               2000 PRINT A
.                             .
.                             .
```

The value printed for A depends upon whether or not the first program has reached its line 2000 yet. If it hasn't, then A is still 0, otherwise it has been changed. But the execution time of the first program varies with K and so we cannot be sure what effect the program will have.

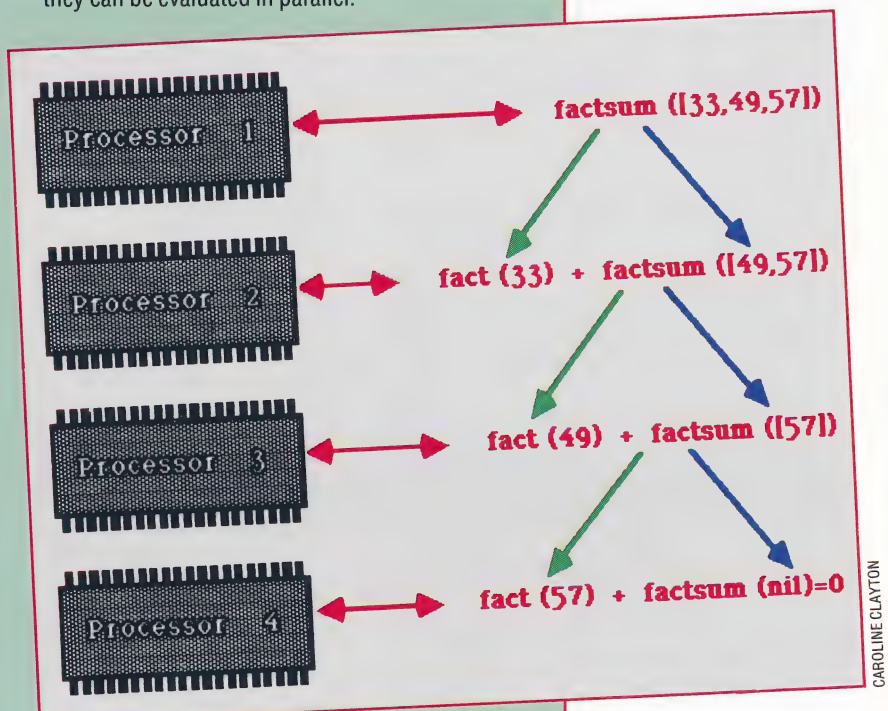
In a language like HOPE, with no assignment to variables, this problem does not arise. Let's take as an example a program to compute the sum of the factorials of a list of numbers. We have already seen the factorial function:

```
    dec fact : num —> num;
--- fact (0)          <= 1;
--- fact (succ(n))  <= (succ(n) * fact (n));
```

Now we can write factsum using fact:

```
    dec factsum : list (num) —> num:
--- factsum (nil)  <= 0;
--- factsum (x :: y) <= fact (x) + factsum (y) ;
```

This says that the factsum of an empty list is zero, otherwise it is formed by adding the factorial of the first element to the factsum of the rest. The two terms on the right-hand side of the second equation are completely independent of one another, and so they can be evaluated in parallel.



CAROLINE CLAYTON

At each stage, a new processor can be brought in to calculate the fact term, and the first processor adds up all the partial results when the recursion 'unwinds' upon reaching 0. The Alice parallel computer being developed at Imperial College executes HOPE in this way, using a ring of Inmos Transputers. Any processor in the ring that is idle can take on one of the partial calculations, which are put on 'offer' by circulating their descriptions around the ring

ECONOMY SIZE

Following our overview of methods of text compression in the previous instalment, we now proceed to develop one of these techniques into a general purpose text compression program. Our implementation, which is divided into two sections, uses a four-bit compression algorithm.

The method of compression we are using in our program is a cut-down implementation of the four-bit compression algorithm described in the previous instalment (see page 1834). It works by translating text into four-bit blocks, each of which is either a commonly used character, or else a code giving information about the four bits that follow. The significance of the different four-bit codes is shown in the Nibble Values diagram.

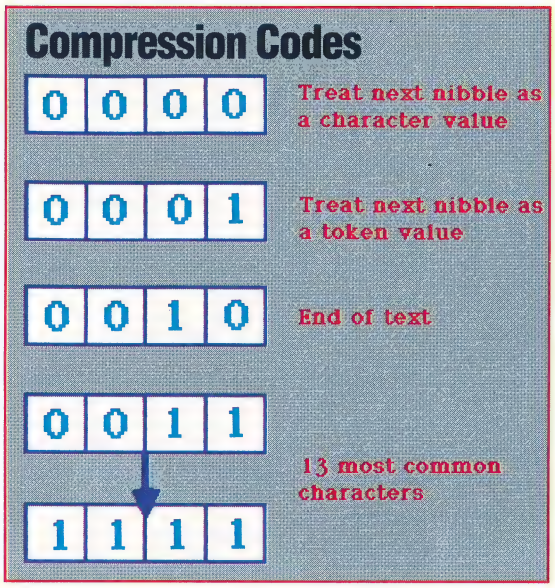
As it stands, the program will only compress text consisting of upper case letters, spaces, commas and full stops. It will also support tokens, and 16 of these have been included in the program to illustrate this feature.

The program requires the user to specify the address of the input string and the address of where the output string is to be placed. The string is stored as a count byte, followed by the bytes of the string. String length is limited to 255 bytes.

The program is fairly easy to modify, so machine code programmers should have no trouble in turning it into a full implementation. The most effective modification would be to include a facility for referencing a 255-element token table. This would enable you to include both upper and lower case characters as well as more tokens for common words. This facility could be easily introduced by allocating a nibble value of three to mean 'treat the next eight bits as an offset value into a 255-element table'. This should be inserted before the check for eight-bit tokens and will essentially be the same as the existing routine, except for the table it addresses.

If you were to introduce this modification, you should rearrange the tables so that the four-bit character values represented the most common *lower case* letters (plus space and newline characters). The next most common lower case letters go into the eight-bit table, and the least common letters (plus upper case and tokens) are placed in the new table.

The ORG address of the program may be changed to suit different micros. The program should be assembled and the object code stored on tape or disk in preparation for the next instalment, when we will give the BASIC program to drive the utility.



Nibble Values
Our program encodes text into sequences of four bits (nibbles), which have the significance shown in the diagram. Note that only values 0 to 2 are used as 'signals'. The programmer could, however, include the value 3 as a signal that the following two nibbles should be taken to indicate an offset in a 255-element table, thereby allowing lower case characters and more tokens to be included

Text Compressor Program

Our text compression program is given in two parts. Here we present an assembly listing for Z80 machines, and in the next instalment we will provide a 6502 version. BASIC loaders, for readers who do not possess assemblers, will also be provided for both implementations

```
        org 30000

        jr      start           ; leap round the data space

string: dw      0       ; load with input string address
output: dw      0       ; enter address for output string
status: db      0       ; status return, zero signals ok
mask:   db      0       ;
len:    db      0       ; storage space for input length

start:  ld      hl,(string)     ; get address of string to
                                ; compress
        ld      a,(hl)          ; get input string length
        ld      (len),a         ; store it away
        inc     hl              ; point at start of string
        ld      de,(output)     ; point at output space
        push    de              ; save start of output
        inc     de              ; leave 1 space for count
        ld      (output),de     ; save output address till
                                ; we get something to store
        ld      a,255
        ld      (mask),a        ; handle left nibble first

nchar:  ld      a,(hl)          ; get byte to work on
        call    check           ; make sure it's within range
        jr      nz,badchar      ; jr if character not allowed
        call    token           ; check if start of a token
        jr      z,gottoken      ; found a token so process it
        call    fourbit         ; check for a 4 bit character
        jr      z,got4bit       ; character is 4 bits so jump
        call    eightbit        ; get 8 bit character value
        push    af              ; no need to check - save value
        ld      a,0             ; while '8 bit char' indicator..
        call    writenib        ;..output
```

```
        pop     af              ; restore 2nd nibble of 8 bit val
got4bit:call    writenib        ; output 2nd nibble of 8 bit, or
                                ; 1st nibble of 4 bit, value
rejoin: inc     hl              ; point at next input character
        ld      a,(len)         ; get back current length
        dec     a               ; minus one for char just proc'd
        ld      (len),a         ; put back new length
        and     a
        jr      nz,nchar        ; jr to proc next char if exists
        ld      (status),a      ; signal finished ok, a will = 0
        ld      a,2             ; 2 = end of compressed text
        call    writenib        ; write it out
        ex      de,hl           ; final output address into hl
        pop     de              ; get output string address
        ld      a,(mask)        ; get mask value
        and     a
        jr      z,nodec         ; if mask zero then final value ok
        dec     hl              ; ignore final byte, it's null
nodec:  and     a               ; clear carry for subtraction
        sbc     hl,de           ; get length
        ld      a,l             ; get length - not more than 255!
        ld      (de),a          ; store count at begin of output
        ret                     ; return

badchar:pop     de              ; clear stack
        ld      a,255
        ld      (status),a      ; signal failure
        ret

gottoken:
        push    af              ; save token nibble
        ld      a,1             ; signal token coming
        call    writenib
        pop     af              ; restore nibble
        call    writenib        ; output token
        jr      rejoin

        ; subroutine to check for start of new token
token:  push    hl              ; save input address
        ex      de,hl           ; put input address into de
        ld      hl,tktable      ; point hl at table of tokens
        ld      a,(len)         ; get remaining length into a
        ld      c,a             ; and transfer it to c
        ld      b,0fh           ; initialize token number
tok1:   ld      a,(hl)          ; get length of current token
        and     a
        jr      z,notfound      ; if length=zero then endoftable
        push    de              ; save these pointers in case
        push    hl              ; they are needed
        inc     hl              ; point to 1st char in token
        push    bc              ; save token number
        ld      b,a             ; count into b
tok2:   ld      a,(de)          ; get input character
        cp      (hl)            ; compare with token character
        jr      nz,nxttoken     ; if no match forget this token
        inc     hl              ; point to next character
        inc     de
        dec     b               ; reduce remaining chars count
        jr      nz,chkchars     ; if b isn't 0 then jump
        ld      a,c
        ld      (len),a         ; store it away
        pop     bc              ; token found - restore token no.
        pop     hl              ; clear stack
        pop     hl              ;
        pop     hl              ;
        ex      de,hl           ; put new input address into hl
        dec     hl              ; and dec to keep main loop in step
        xor     a               ; set zero flag to signal success
        ld      a,b             ; get token number into a
        ret
chkchars:
        dec     c               ; reduce remaining character count
        jr      nz,tok2         ; if still chars to check go back
nxttoken:
        pop     bc              ; restore remaining chars+tok no.
        dec     b               ; next token number
        pop     hl              ; get back pointer to curr tok. no
        ld      a,(hl)
        ld      e,a             ; length into de
        ld      d,0
        inc     hl              ; step over count
        add     hl,de           ; hl now point at next tokens count
        pop     de              ; restore input address
        jr      tok1            ; jump to check next token
notfound:
        pop     hl              ; restore original input address
        ld      a,(hl)          ; restore character
        or      a               ; reset zero flag to signal failure
        ret

        ; subroutine to check for 4 bit characters

fourbit:push    hl              ; save input address
        ld      hl,tab4bit      ; point hl at character table
        call    tabscan         ; scan table for value
        pop     hl              ; restore input address
        ret                     ; go back

        ; subroutine to check for 8 bit characters
eightbit:
        push    hl              ; save input address
        ld      hl,tab8bit      ; point at 8 bit character table
        call    tabscan         ; scan table
        pop     hl              ; restore address
        ret                     ; return

        ; general purpose table scanning routine

tabscan:ld      b,0fh           ; 16 characters in each table
                                ; (one is numbered 0)
tabsc2: cp      (hl)            ; check character
        jr      z,tabsc3        ; return with 0 set if match
        inc     hl              ; check next table element
        djnz    tabsc2          ; loop till no chars in table
        or      a               ; reset 0 flag to signal no match
                                ; a contains char code <> 0
        ret                     ; go back
tabsc3: ld      a,b             ; put nibble into a
        ret

        ; routine to check for valid characters
        ; only space  , . ; : and upper case letters allowed
check:  cp      ' '             ; check for space first
        ret     z
        cp      ','             ; amd comma
        ret     z
        cp      '.'             ; finally full stop
        ret     z
        cp      'A'             ; assume contiguous alphebet
        jr      c,nogood        ; jump if 'less than' A
        cp      'Z'+1
        jr      nc, nogood
        ld      c,a
        xor     a               ; set zero to signal character ok
        ld      a,c
        ret
nogood: ld      a,255
        and     a               ; reset zero to signal failure
        ret

writenib:
        ld      c,a             ; save nibble
        ld      a,(mask)        ;
        ld      de,(output)     ; get address of byte to write to
        and     a               ; check mask value
        jr      nz,left         ; jump if nibble needs shifting
        ld      a,(de)          ; get old nibble
        or      c               ; insert new nibble
        ld      (de),a          ; and store it back
        inc     de              ; point to next byte
        ld      (output),de     ; save address
        ld      a,255
        ld      (mask),a        ; signal left nibble next time
        ret
left:   ld      a,c             ; get value into a
        sla     a                       ; shift it across
        sla     a
        sla     a
        sla     a
        ld      (de),a          ; amd store it
        xor     a
        ld      (mask),a        ; signal other nibble next time
```

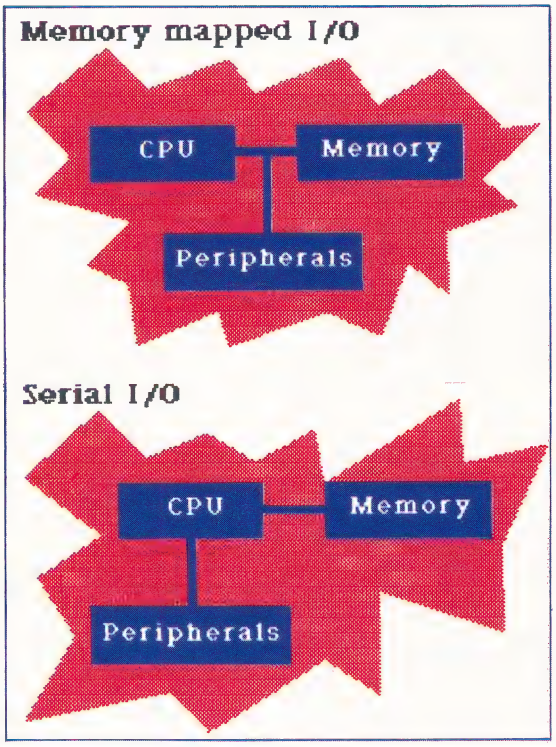
```
        ret

; 4 Bit text expansion program for Z80, machine independent

expand: ld      hl,(string)     ; string contains address of
                                ; string to expand
        inc     hl              ; point hl at start of string
        ld      (string),hl     ; put back new string start
        ld      de,(output)     ; point at start of output space
        push    de              ; save for end
        ld      a,255
        ld      (mask),a        ; handle left nibble first

nextnib:call getnib             ; get next input nibble
        cp      2               ; is this the end
        jp      z,fin           ; if so jump to finish up
        jp      nc,exp4bit      ; if nibble > 2 process 4 bits
        and     a
        jp      z,exp8bit       ; if nibble=1 process 8 bit char
        call    getnib          ; get nibble to expand into token
        call    tokscan         ; find the one we want
tokloop:ld      b,a             ; count into b
tklp1:  ld      a,(hl)          ; get token character
        inc     hl              ; point to next character
        call    outchar         ; output to new string
        djnz    tklp1           ; loop for all the token chars
        jr      nextnib         ; jump back for next input nibble
tokscan:ld      hl,tktable
        ld      b,a             ; initialise b
        ld      a,0fh
        sub     b
        ld      b,a             ; count into b
        inc     b
toksc1: ld      a,(hl)          ; get lenght of this token
        inc     hl              ; point at first char in token
        dec     b               ; reduce count
        ret     z               ; return if it's the one we want
        ld      e,a             ; step over this token
        ld      d,0
        add     hl,de
        jr      toksc1          ; jump back

fin:    ld      hl,(output)     ; get end of output string in hl
        pop     de              ; start of the output string
        and     a               ; clear carry
        sbc     hl,de           ; length in hl
        ld      a,l             ; length mustn't be > 255
        ld      (de),a          ; put count in right place
        ret                     ; return to system

exp8bit:call    getnib          ; get next nibble for 8 bit char
        ld      hl,tab8bit      ; point hl at correct table
        call    index           ; get the right character
        call    outchar         ; put it in the output string
        jr      nextnib         ; go back for more

exp4bit:ld      hl,tab4bit      ; point at 4 bit table
        call    index           ; find character
        call    outchar         ; output it
        jr      nextnib         ; round again

index:  ld      e,a             ; initialise e
        ld      a,0fh
        sub     e               ; get offset into a
        ld      e,a             ; put into e
        ld      d,0
        add     hl,de           ; index into table
        ld      a,(hl)          ; get character
        ret

outchar:ld      de,(output)     ; get last address outputted to
        inc     de              ; point to new address
        ld      (de),a          ; store character
        ld      (output),de     ; put back new address
        ret

getnib: ld      a,(mask)        ; get mask value
        ld      hl,(string)     ; get address of input byte
        and     a
        ld      a,(hl)          ; get byte
        jr      nz,shfta        ; jump if left nibble required
        and     0fh             ; mask right nibble
        inc     hl              ; point to new byte
        ld      (string),hl     ; store for next time
        ld      c,a
        ld      a,255           ; signal left nibble next time
        ld      (mask),a
        ld      a,c
        ret
shfta:  sra     a               ; shift right nibble ove
        sra     a
        sra     a
        sra     a
        and     0fh             ; mask it off
        ld      c,a
        xor     a               ; signal right nibble next time
        ld      (mask),a
        ld      a,c
        ret

        ; table of values

        ; four bit table

tab4bit:db      ' '             ; space character
        db      'E'
        db      'T'
        db      'A'
        db      'O'
        db      'N'
        db      'R'
        db      'I'
        db      'S'
        db      'H'
        db      'D'
        db      'L'
        db      'F'
        db      0               ; dummy values to fill table
        db      0
        db      0

        ; 8 bit characters

tab8bit:db      'C'
        db      'M'
        db      'U'
        db      'G'
        db      'Y'
        db      'P'
        db      'W'
        db      'B'
        db      'V'
        db      'K'
        db      'X'
        db      'J'
        db      'Q'
        db      'Z'
        db      ','
        db      '.'
        ; token table
tktable:db      3,'THE'
        db      4,'THIS'
        db      4,'THAT'
        db      2,'IF'
        db      3,'YOU'
        db      2,'ME'
        db      3,'WAS'
        db      2,'HE'
        db      3,'SHE'
        db      4,'THEY'
        db      2,'OF'
        db      2,'IT'
        db      2,'IS'
        db      3,'FOR'
        db      2,'ON'
        db      2,'TO'
        db      0               ; end of table marker

        end
```

A BIT AT A TIME

On the Motorola 68000 processor, peripherals are connected to the same communication bus as the memory and CPU. In this instalment, we examine this 'memory mapped' form of I/O communication, and see what information is required by the CPU to make memory mapping as efficient as possible.

There are two basic methods used to select and communicate with I/O devices:

- *Memory mapped:* This system shares the central communicating bus (to which the CPU and the memory are connected) with the I/O devices. The peripherals are merely added on to this bus via appropriate electronics. This means that the peripheral becomes memory mapped, since the processor can now read and write to the device just like ordinary memory. This system pays, however, by losing speed on the bus, since each peripheral is competing for service.
- *Isolated I/O:* In this system, there is a separate bus dedicated to I/O devices, so the speed of data transfer can increase compared to the memory mapped I/O. But you lose out again from the standpoint of convenience, because there are now separate I/O instructions dedicated to the peripherals.



CAROLINE CLAYTON

Bus Routes
Our diagram shows the difference between serial I/O, which uses a dedicated peripheral bus to exchange information, and memory mapped I/O, where peripherals take data from the same bus as that used to access memory. In the latter case, certain addresses will need to be reserved for peripherals, which the computer may then access in a similar way to accessing RAM or ROM

It's not surprising at this point, then, to learn that the 68000 has memory mapped I/O. Earlier on in this series (see page 1697), we looked at the microcomputer structure. We saw that some peripheral devices were connected to the same bus as the memory and CPU, which would indicate mapped I/O.

Let's now look at the three main types of information the processor requires, which is in this case what is mapped into the memory.

- *Status:* Since the peripherals will take a finite amount of time to complete the desired operation — print a character, for instance, or read a keyboard — we need status information to prevent the peripheral from being commanded before it has finished the last operation. Other information is also required that is not directly concerned with the operation but may be concerned with, say, the mode of operation or any failure condition, such as out of paper on a printer.
- *Control:* We need a register that will enable us to configure or actuate the peripheral we are using — to command a disk device to read a block of data, for example, or configure a communication channel for a specific baud rate.
- *Data:* Finally, we need to be able to read or write the data to the peripheral and hold that data there until the operation has been completed, ready to receive new information.

I/O CHIPS

At the level of signals on the main computer bus, the detailed operation of interface chips can be quite complicated. This complication is removed (from the programmer's point of view) when special purpose interface chips are provided. Motorola provides two such chips, one for serial device control (ACIA) and the other for parallel control (PIA).

Serial devices will be connected to the chip using only two wires to send and two wires to receive the information; the bits that make up the data bytes come one after the other. For the parallel case, the bits come a byte at a time, all at the same time. The peripheral itself is connected to the chip with a byte (or even a word) number of lines.

One of the advantages of using these interface chips is that the program has a high degree of flexibility in setting the hardware configuration details. However, this means that the number of bits in the control word, in particular, can be quite large. For example, one ACIA control bit assignment is:

Bits 0 and 1	clock division frequency and initialisation (serial bit rate)
Bits 2 to 4	serial character format (such as parity or not, number of stop bits)
Bits 5 and 6	transmitter control bits (such as interrupt enable)
Bit 7	receiver control bit set for interrupt enable

The order in which these bits are set up must primarily ensure that sensible communication can take place between the computer and the peripheral. Failure to do this can lead to complete nonsense at the receiving device.

Two instructions are required for any particular serial I/O. One of these configures the ACIA to the serial peripheral. For example:

```
MOVE.B   #$3,ACIACON    reset the hardware
                        initially
MOVE.B   #$15,ACIACON   configure the hardware
```

In this case, ACIACON will have previously been defined.

Similarly, there will be status bits that have to be read for correct data transfers. For instance:

Bit 0	receive data register ready (a new character is available)
Bit 1	transmit register empty (a new character can be sent)
Bits 2 and 3	modem control signals (CTS and DCD)
Bits 4 to 6	error indications on received data (such as a parity error)
Bit 7	interrupt request

The interrupts will be dealt with in the next article, but the modems are outside the scope of this series.

Since we only ever read from a status register and write to a control register, we could use only one address to achieve the memory mapping. This also applies to the data register, since we will only ever read from the receive data register and write to the transmit register.

As an example, if we need to input a character,

```
INCH  BTST    #0, ACIASTAT   is there a character
                             available?
      BEQ     INCH           keep testing if not
      MOVE.B  ACIADATA, DO   read char into DO as
                             a parameter
      RTS                    return from
                             subroutine
```

In this case, we wait until bit 0 is set in the control register before reading from the data register. ACIASTAT equals ACIACON in this example.

Similarly, we can also write a subroutine to output characters:

```
OUTCH BTST    #1ACIASTAT     wait until data
                             register is empty
      BEQ     OUTCH          if not, branch
                             and test again
      MOVE.B  DO,ACIADATA    output the
                             character
      RTS
```

If we needed to repeat any received character, we'd simply call one subroutine after the other:

```
ECHO  JSR     INCH           read a character
      JSR     OUTCH          output the
                             character
      BRA     ECHO           and repeat
```

It should be emphasised here that performing I/O in this manner is very inefficient because the computer will be working at the rate of character transmission. But it nevertheless serves to illustrate the basic mechanisms of programmed input/output. In the next instalment, we'll see how interrupts help the situation and see how to perform parallel I/O data transfers.

68000 Instruction Summary

Before we move on to look at the interrupt facilities of the 68000, we summarise here all the 68000 instructions we have examined so far in this series. First, the data copying group which has the MOVE instruction as its focus:

Instruction	Operation
MOVE	Move source to destination
MOVEM	Move address and data registers to/from memory (Useful on entry and exit to subroutines)
MOVEA	Move contents of effective address to address register
LEA	Move source address to address register
MOVEQ	Quickly move small constant to data register
PEA	Push effective address onto stack
SWAP	Swap upper and lower words in a data register
EXG	Exchange longwords between certain registers
LINK/UNLK	Used to frame parameters for subroutine calls (specialised)

This set of instructions provides some very comprehensive facilities, but you should be careful how to use the more exotic commands (LINK/UNLK, for example). Now let's look at the instructions dealing with integer arithmetic:

Instruction	Operation
ADD	Add source to destination
ADDA	Add source to destination address register
ADDI	Add immediate data to data alterable destination
ADDQ	Quick immediate add for small constants (1 to 8)
ADDX	Add with a carry
SUB(A/I/Q/X)	Subtract source from destination in any one of the corresponding addressing varieties shown for ADD
CMP(A/I)	Compare source with destination and set condition codes
CMPM	Compare memory locations and post-increment pointers
NEG(X)	Negate operand data register
EXT	Sign extend on the operand data register

MULU	Multiply unsigned data in data register by effective address, result placed in data register
MULS	As for MULU, but for signed data
DIVU(S)	Divide destination data register by source operand. Quotient and remainder held in destination register
TST	Set the condition codes according to the operand
TAS	Test and set most significant bit of operand
CLR	Clear the effective address

Quite a wide-ranging set of instructions, but you need care in selecting the right instruction for the operand addressing modes! There now follows a list of the instructions concerned with BCD arithmetic:

Instruction	Operation
ABCD	Add BCD operands to produce sum in destination
SBCD	Subtract the operands
NBCD	Negate the BCD operand

Note that there is no BCD multiply instruction. However, the logical instructions, which follow, are totally comprehensive:

Instruction	Operation
AND	Logical AND of source with the destination, at least one operand being a data register
ANDI	Logical AND with immediate data as source
OR	Logical OR using same convention as above
ORI	Logical OR with immediate data
EOR	Exclusive OR
EORI	Exclusive OR with immediate data
NOT	Logical invert operand

We now list the instructions concerned with bit testing and manipulation:

BIT MANIPULATION GROUP

Instruction	Operation
BTST	Test the bit specified in the source operand
BSET	Test and set the destination operand
BCLR	Test and clear
BCHG	Test and change the operand

The first group is quite useful — remember that you don't have to take action on the tested bit (i.e. you can just use the instruction to manipulate a bit)

SHIFT AND ROTATE GROUP

Instruction	Operation
ASI	Arithmetic shift left
ASR	Arithmetic shift right
LSL	Logical shift left
LSR	Logical shift right
ROL	Rotate operand left, setting carry appropriately
ROR	Rotate operand right, setting carry

Finally, we look at the program control and subroutine instructions:

PROGRAM CONTROL INSTRUCTIONS

Instruction	Operation
BRA	Branch always (an efficient form of unconditional branch)
JMP	Jump unconditionally (useful if you want to do some form of computation on the jump address)
Bcc	Conditional branch depending on the condition code 'cc' being tested. That is:

For Signed Operands:

GT	greater than
LT	less than
GE	greater than or equal to
LE	less than or equal to
VS	overflow set
VC	overflow clear

For Unsigned Operands:

EQ	equal to
NE	not equal to
MI	minus
PI	plus
HI	higher than
LS	lower than or same
CS	carry set
CC	carry clear

DBcc	Decrement loop counter and branch on condition 'cc', as above. Remember the different sense of the branch instruction though!

SUBROUTINE CONTROL INSTRUCTIONS

Instruction	Operation
JSR	Jump to subroutine
BSR	Efficient form of JSR
RTS	Return from subroutine

The summary here does not include all the 68000 instructions. There is one particular group concerned with system control using privileged instructions that has deliberately been omitted. These instructions would generally be used by operating system designers, and as such are beyond the scope of these articles. The instructions concern operations on the status register and stack pointers, and the 'software interrupt' instructions called 'traps'. Further details can, of course, be obtained from the 68000 user manual

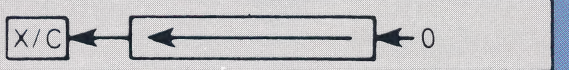
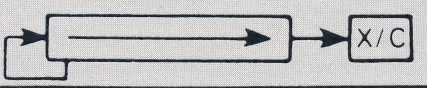
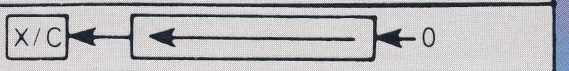
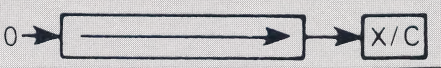
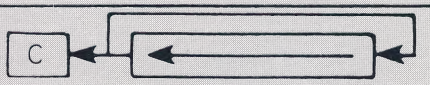
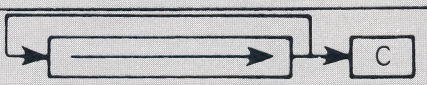
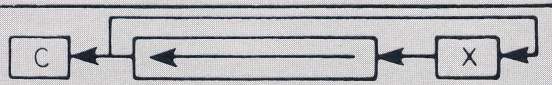
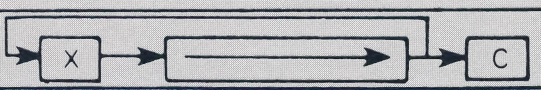
Motorola 68000 Instruction Set

Here, courtesy of Motorola Inc, we present the second of three instalments in which we give details of the 68000's instruction set broken down into its eight component classes

Logical Operations

Instruction	Operand Size	Operation
AND	8, 16, 32	Dn∧(EA) → Dn (EA)∧Dn → (EA) (EA)∧#xxx → (EA)
OR	8, 16, 32	Dn v (EA) → Dn (EA) v Dn → (EA) (EA) v #xxx → (EA)
EOR	8, 16, 32	(EA) ⊕ Dy → (EA) (EA) ⊕ #xxx → (EA)
NOT	8, 16, 32	~(EA) → (EA)

Bit Manipulation Operations

Instruction	Operand Size	Operation
ASL	8, 16, 32	
ASR	8, 16, 32	
LSL	8, 16, 32	
LSR	8, 16, 32	
ROL	8, 16, 32	
ROR	8, 16, 32	
ROXL	8, 16, 32	
ROXR	8, 16, 32	

NOTES:
~ = invert
# = immediate data
∧ = logical AND
V = logical OR
⊕ = logical exclusive OR

Shift And Rotate Operations

Instruction	Operand Size	Operation
BTST	8, 32	~bit of (EA) → Z
BSET	8, 32	~bit of (EA) → Z 1 → bit of EA
BCLR	8, 32	~bit of (EA) → Z 0 → bit of EA
BCHG	8, 32	~bit of (EA) → Z ~bit of (EA) → bit of EA

NOTE ~ = invert


© 1983 COLLERY, M.—CRANSTON–CSURI